下载APP

16｜复合数据类型：原生map类型的实现机制是怎样的？

2021-11-17 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 22:53 大小 20.97M

你好，我是 Tony Bai。

上一节课，我们学习了 Go 语言中最常用的两个复合类型：数组与切片。它们代表**一组连续存储的同构类型元素集合。**不同的是，数组的长度是确定的，而切片，我们可以理解为一种“动态数组”，它的长度在运行时是可变的。

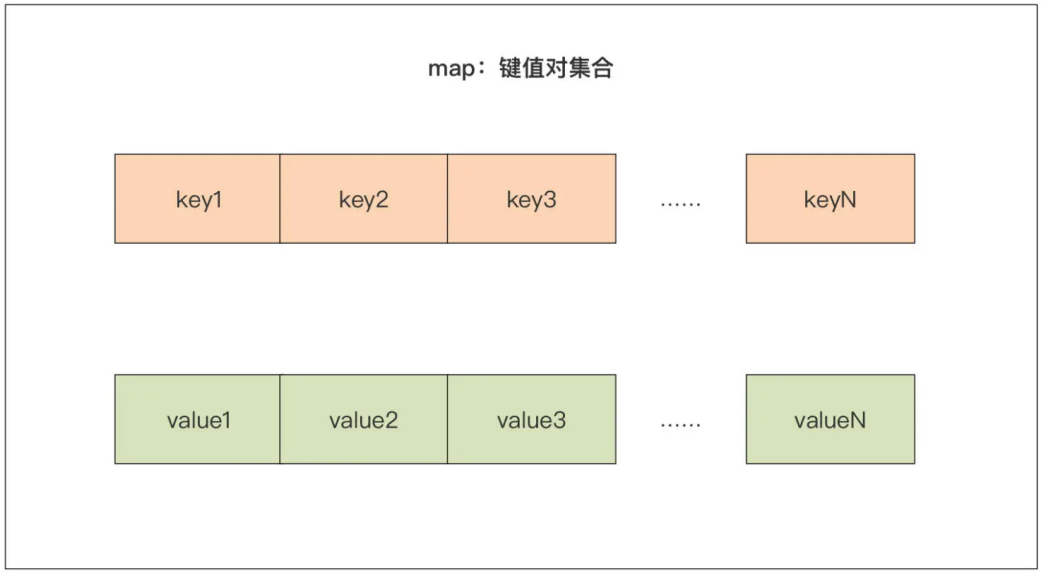
领资料

这一节课，我们会继续前面的脉络，学习另外一种日常 Go 编码中比较常用的复合类型，这种类型可以让你将一个值（Value）唯一关联到一个特定的键（Key）上，可以用于实现特定键值的快速查找与更新，这个复合数据类型就是 **map**。很多中文 Go 编程语言类书籍都会将它翻译为映射、哈希表或字典，但在我的课程中，**为了保持原汁原味，我就直接使用它的英文名，map。**

map 是我们既切片之后，学到的第二个由 Go 编译器与运行时联合实现的复合数据类型，它有着复杂的内部实现，但却提供了十分简单友好的开发者使用接口。这一节课，我将从 map 类型的定义，到它的使用，再到 map 内部实现机制，由浅到深地让你吃透 map 类型。

什么是 map 类型？

map 是 Go 语言提供的一种抽象数据类型，它表示一组无序的键值对。在后面的讲解中，我们会直接使用 key 和 value 分别代表 map 的键和值。而且，map 集合中每个 key 都是唯一的：



极客时间

和切片类似，作为复合类型的 map，它在 Go 中的类型表示也是由 key 类型与 value 类型组成的，就像下面代码：

复制代码

```
map[key_type]value_type
```

key 与 value 的类型可以相同，也可以不同：

复制代码

```
map[string]string // key与value元素的类型相同
map[int]string    // key与value元素的类型不同
```

如果两个 map 类型的 key 元素类型相同，value 元素类型也相同，那么我们可以说它们是同一个 map 类型，否则就是不同的 map 类型。

这里，我们要注意，map 类型对 value 的类型没有限制，但是对 key 的类型却有严格要求，因为 map 类型要保证 key 的唯一性。Go 语言中要求，**key 的类型必须支持“==”和“!=”两种比较操作符。**

但是，在 Go 语言中，函数类型、map 类型自身，以及切片只支持与 nil 的比较，而不支持同类型两个变量的比较。如果像下面代码这样，进行这些类型的比较，Go 编译器将会报错：

复制代码

```
s1 := make([]int, 1)
s2 := make([]int, 2)
f1 := func() {}
f2 := func() {}
m1 := make(map[int]string)
m2 := make(map[int]string)
println(s1 == s2) // 错误：invalid operation: s1 == s2 (slice can only be compa
println(f1 == f2) // 错误：invalid operation: f1 == f2 (func can only be compar
println(m1 == m2) // 错误：invalid operation: m1 == m2 (map can only be compare
```

因此在这里，你一定要注意：**函数类型、map 类型自身，以及切片类型是不能作为 map 的 key 类型的。**

知道如何表示一个 map 类型后，接下来，我们来看看如何声明和初始化一个 map 类型的变量。

map 变量的声明和初始化

我们可以这样声明一个 map 变量：

复制代码

```
var m map[string]int // 一个map[string]int类型的变量
```

和切片类型变量一样，如果我们没有显式地赋予 map 变量初值，map 类型变量的默认值为 nil。

不过切片变量和 map 变量在这里也有些不同。初值为零值 nil 的切片类型变量，可以借助内置的 append 的函数进行操作，这种在 Go 语言中被称为“**零值可用**”。定义“零值可用”的类型，可以提升我们开发者的使用体验，我们不用再担心变量的初始状态是否有效。

但 map 类型，因为它内部实现的复杂性，无法“零值可用”。所以，如果我们对处于零值状态的 map 变量直接进行操作，就会导致运行时异常（panic），从而导致程序进程异常退出：

复制代码

```
var m map[string]int // m = nil
m["key"] = 1         // 发生运行时异常：panic: assignment to entry in nil map
```

所以，我们必须对 map 类型变量进行显式初始化后才能使用。那我们怎样对 map 类型变量进行初始化呢？

和切片一样，为 map 类型变量显式赋值有两种方式：一种是使用复合字面值；另外一种是使用 make 这个预声明的内置函数。

方法一：使用复合字面值初始化 map 类型变量。

我们先来看这句代码：

复制代码

```
m := map[int]string{}
```

这里，我们显式初始化了 map 类型变量 m。不过，你要注意，虽然此时 map 类型变量 m 中没有任何键值对，但变量 m 也不等同于初值为 nil 的 map 变量。这个时候，我们对 m 进行键值对的插入操作，不会引发运行时异常。

这里我们再看看怎么通过稍微复杂一些的复合字面值，对 map 类型变量进行初始化：

复制代码

```
m1 := map[int][]string{
    1: []string{"val1_1", "val1_2"},
    3: []string{"val3_1", "val3_2", "val3_3"},
    7: []string{"val7_1"},
}

type Position struct {
    x float64
    y float64
}

m2 := map[Position]string{
    Position{29.935523, 52.568915}: "school",
    Position{25.352594, 113.304361}: "shopping-mall",
    Position{73.224455, 111.804306}: "hospital",
}
```

我们看到，上面代码虽然完成了对两个 map 类型变量 m1 和 m2 的显式初始化，但不知道你有没有发现一个问题，作为初值的字面值似乎有些“臃肿”。你看，作为初值的字面值采用了复合类型的元素类型，而且在编写字面值时还带上了各自的元素类型，比如作为 map[int] []string 值类型的[]string，以及作为 map[Position]string 的 key 类型的 Position。

别急！针对这种情况，Go 提供了“语法糖”。这种情况下，**Go 允许省略字面值中的元素类型**。因为 map 类型表示中包含了 key 和 value 的元素类型，Go 编译器已经有足够的信息，来推导出字面值中各个值的类型了。我们以 m2 为例，这里的显式初始化代码和上面变量 m2 的初始化代码是等价的：

复制代码

```
m2 := map[Position]string{
    {29.935523, 52.568915}: "school",
    {25.352594, 113.304361}: "shopping-mall",
    {73.224455, 111.804306}: "hospital",
}
```

以后在无特殊说明的情况下，我们都将使用这种简化后的字面值初始化方式。

方法二：使用 make 为 map 类型变量进行显式初始化。

和切片通过 make 进行初始化一样，通过 make 的初始化方式，我们可以为 map 类型变量指定键值对的初始容量，但无法进行具体的键值对赋值，就像下面代码这样：

复制代码

```
m1 := make(map[int]string) // 未指定初始容量
m2 := make(map[int]string, 8) // 指定初始容量为8
```

不过，map 类型的容量不会受限于它的初始容量值，当其中的键值对数量超过初始容量后，Go 运行时会自动增加 map 类型的容量，保证后续键值对的正常插入。

了解完 map 类型变量的声明与初始化后，我们就来看看，在日常开发中，map 类型都有哪些基本操作和注意事项。

map 的基本操作

针对一个 map 类型变量，我们可以进行诸如插入新键值对、获取当前键值对数量、查找特定键和读取对应值、删除键值对，以及遍历键值等操作。我们一个个来学习。

操作一：插入新键值对。

面对一个非 nil 的 map 类型变量，我们可以在其中插入符合 map 类型定义的任意新键值对。插入新键值对的方式很简单，我们只需要把 value 赋值给 map 中对应的 key 就可以了：

复制代码

```
m := make(map[int]string)
m[1] = "value1"
m[2] = "value2"
m[3] = "value3"
```

而且，我们不需要自己判断数据有没有插入成功，因为 Go 会保证插入总是成功的。这里，Go 运行时会负责 map 变量内部的内存管理，因此除非是系统内存耗尽，我们可以不用担心向 map 中插入新数据的数量和执行结果。

不过，如果我们插入新键值对的时候，某个 key 已经存在于 map 中了，那我们的插入操作就会用新值覆盖旧值：

复制代码

```
m := map[string]int {
  "key1" : 1,
  "key2" : 2,
}

m["key1"] = 11 // 11会覆盖掉"key1"对应的旧值1
m["key3"] = 3  // 此时m为map[key1:11 key2:2 key3:3]
```

从这段代码中你可以看到，map 类型变量 m 在声明的同时就做了初始化，它的内部建立了两个键值对，其中就包含键 key1。所以后面我们再给键 key1 进行赋值时，Go 不会重新创建 key1 键，而是会用新值 (11) 把 key1 键对应的旧值 (1) 替换掉。

操作二：获取键值对数量。

如果我们在编码中，想知道当前 map 类型变量中已经建立了多少个键值对，那我们可以怎么做呢？和切片一样，map 类型也可以通过内置函数 **len**，获取当前变量已经存储的键值对数量：

复制代码

```
m := map[string]int {
  "key1" : 1,
  "key2" : 2,
}

fmt.Println(len(m)) // 2
m["key3"] = 3
fmt.Println(len(m)) // 3
```

不过，这里要注意的是**我们不能对 map 类型变量调用 cap，来获取当前容量**，这是 map 类型与切片类型的一个不同点。

操作三：查找和数据读取

和写入相比，map 类型更多用在查找和数据读取场合。所谓查找，就是判断某个 key 是否存在于某个 map 中。有了前面向 map 插入键值对的基础，我们可能自然而然地想到，可以用下面代码去查找一个键并获得该键对应的值：

复制代码

```
1 m := make(map[string]int)
2 v := m["key1"]
```

乍一看，第二行代码在语法上好像并没有什么不当之处，但其实通过这行语句，我们还是无法确定键 key1 是否真实存在于 map 中。这是因为，当我们尝试去获取一个键对应的值的时候，如果这个键在 map 中并不存在，我们也会得到一个值，这个值是 value 元素类型的**零值**。

我们以上面这个代码为例，如果键 key1 在 map 中并不存在，那么 v 的值就会被赋予 value 元素类型 int 的零值，也就是 0。所以我们无法通过 v 值判断出，究竟是因为 key1 不存在返回的零值，还是因为 key1 本身对应的 value 就是 0。

那么在 map 中查找 key 的正确姿势是什么呢？Go 语言的 map 类型支持通过用一种名为“**comma ok**”的惯用法，进行对某个 key 的查询。接下来我们就用“comma ok”惯用法改造一下上面的代码：

复制代码

```
1 m := make(map[string]int)
2 v, ok := m["key1"]
3 if !ok {
4     // "key1"不在map中
5 }
6
7 // "key1"在map中，v将被赋予"key1"键对应的value
```

我们看到，这里我们通过了一个布尔类型变量 ok，来判断键“key1”是否存在于 map 中。如果存在，变量 v 就会被正确地赋值为键“key1”对应的 value。

不过，如果我们并不关心某个键对应的 value，而只关心某个键是否在于 map 中，我们可以使用空标识符替代变量 v，忽略可能返回的 value：

复制代码

```
m := make(map[string]int)
_, ok := m["key1"]
... ...
```

因此，你一定要记住：**在 Go 语言中，请使用“comma ok”惯用法对 map 进行键查找和键值读取操作。**

操作四：删除数据。

接下来，我们再看看看如何从 map 中删除某个键值对。在 Go 中，我们需要借助**内置函数 delete** 来从 map 中删除数据。使用 delete 函数的情况下，传入的第一个参数是我们的 map 类型变量，第二个参数就是我们想要删除的键。我们可以看看这个代码示例：

复制代码

```
m := map[string]int {
  "key1" : 1,
  "key2" : 2,
}

fmt.Println(m) // map[key1:1 key2:2]
delete(m, "key2") // 删除"key2"
fmt.Println(m) // map[key1:1]
```

这里要注意的是，**delete 函数是从 map 中删除键的唯一方法**。即便传给 delete 的键在 map 中并不存在，delete 函数的执行也不会失败，更不会抛出运行时的异常。

操作五：遍历 map 中的键值数据

最后，我们来说一下如何遍历 map 中的键值数据。这一点虽然不像查询和读取操作那么常见，但日常开发中我们还是有这个需求的。在 Go 中，遍历 map 的键值对只有一种方法，那就是**像对待切片那样通过 for range 语句对 map 数据进行遍历**。我们看一个例子：

复制代码

```
package main

import "fmt"

```

```
func main() {
    m := map[int]int{
        1: 11,
        2: 12,
        3: 13,
    }

    fmt.Printf("{ ")
    for k, v := range m {
        fmt.Printf("[%d, %d] ", k, v)
    }
    fmt.Printf("}\n")
}
```

你看，通过 for range 遍历 map 变量 m，每次迭代都会返回一个键值对，其中键存在于变量 k 中，它对应的值存储在变量 v 中。我们可以运行一下这段代码，可以得到符合我们预期的结果：

复制代码

```
{ [1, 11] [2, 12] [3, 13] }
```

如果我们只关心每次迭代的键，我们可以使用下面的方式对 map 进行遍历：

复制代码

```
for k, _ := range m {
   // 使用k
}
```

当然更地道的方式是这样的：

复制代码

```
for k := range m {
   // 使用k
}
```

如果我们只关心每次迭代返回的键所对应的 value，我们同样可以通过空标识符替代变量 k，就像下面这样：

复制代码

```
for _, v := range m {
  // 使用v
}
```

不过，前面 map 遍历的输出结果都非常理想，给我们的表象好像是迭代器按照 map 中元素的插入次序逐一遍历。那事实是不是这样呢？我们再来试试，多遍历几次这个 map 看看。

我们先来改造一下代码：

复制代码

```
package main

import "fmt"

func doIteration(m map[int]int) {
    fmt.Printf("{ ")
    for k, v := range m {
        fmt.Printf("[%d, %d] ", k, v)
    }
    fmt.Printf("}\n")
}

func main() {
    m := map[int]int{
        1: 11,
        2: 12,
        3: 13,
    }

    for i := 0; i < 3; i++ {
        doIteration(m)
    }
}
```

运行一下上述代码，我们可以得到这样结果：

复制代码

```
{ [3, 13] [1, 11] [2, 12] }
{ [1, 11] [2, 12] [3, 13] }
{ [3, 13] [1, 11] [2, 12] }
```

我们可以看到，**对同一 map 做多次遍历的时候，每次遍历元素的次序都不相同**。这是 Go 语言 map 类型的一个重要特点，也是很容易让 Go 初学者掉入坑中的一个地方。所以这里你一定要记住：**程序逻辑千万不要依赖遍历 map 所得到的的元素次序**。

从我们前面的讲解，你应该也感受到了，map 类型非常好用，那么，我们在各个函数方法间传递 map 变量会不会有很大开销呢？

map 变量的传递开销

其实你不用担心开销的问题。

和切片类型一样，map 也是引用类型。这就意味着 map 类型变量作为参数被传递给函数或方法的时候，实质上传递的只是一个**“描述符”**（后面我们再讲这个描述符究竟是什么），而不是整个 map 的数据拷贝，所以这个传递的开销是固定的，而且也很小。

并且，当 map 变量被传递到函数或方法内部后，我们在函数内部对 map 类型参数的修改在函数外部也是可见的。比如你从这个示例中就可以看到，函数 foo 中对 map 类型变量 m 进行了修改，而这些修改在 foo 函数外也可见。

复制代码

```
package main

import "fmt"

func foo(m map[string]int) {
    m["key1"] = 11
    m["key2"] = 12
}

func main() {
    m := map[string]int{
        "key1": 1,
        "key2": 2,
    }

    fmt.Println(m) // map[key1:1 key2:2]
    foo(m)
    fmt.Println(m) // map[key1:11 key2:12]
}
```

map 的内部实现

和切片相比，map 类型的内部实现要更加复杂。Go 运行时使用一张哈希表来实现抽象的 map 类型。运行时实现了 map 类型操作的所有功能，包括查找、插入、删除等。在编译阶段，Go 编译器会将 Go 语法层面的 map 操作，重写成运行时对应的函数调用。大致的对应关系是这样的：

复制代码

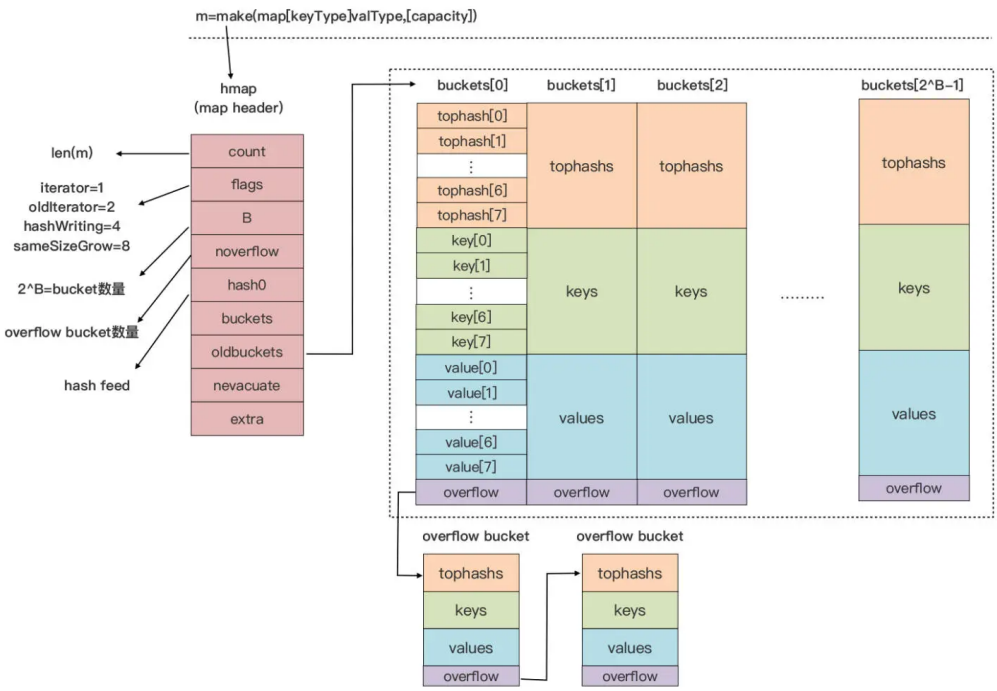
```
// 创建map类型变量实例
m := make(map[keyType]valType, capacityhint) → m := runtime.makemap(maptype, c

// 插入新键值对或给键重新赋值
m["key"] = "value" → v := runtime.mapassign(maptype, m, "key") v是用于后续存储va

// 获取某键的值
v := m["key"]      → v := runtime.mapaccess1(maptype, m, "key")
v, ok := m["key"]  → v, ok := runtime.mapaccess2(maptype, m, "key")

// 删除某键
delete(m, "key")   → runtime.mapdelete(maptype, m, “key”)
```

这是 map 类型在 Go 运行时层实现的示意图：



我们可以看到，和切片的运行时表示图相比，map 的实现示意图显然要复杂得多。接下来，我们结合这张图来简要描述一下 map 在运行时层的实现原理。我们重点讲解一下一个

map 变量在初始状态、进行键值对操作后，以及在并发场景下的 Go 运行时层的实现原理。

初始状态

从图中我们可以看到，与语法层面 map 类型变量（m）一一对应的是 runtime.hmap 的实例。hmap 类型是 map 类型的头部结构（header），也就是我们前面在讲解 map 类型变量传递开销时提到的 **map 类型的描述符**，它存储了后续 map 类型操作所需的所有信息，包括：

count	当前map中的元素个数。对map类型变量运用len内置函数时，len函数返回的就是count这个值
flags	当前map所处的状态标志。目前定义了四个状态值：iterator、oldIterator、hashWriting和sameSizeGrow
B	B的值是bucket数量的以2为底的对数，也就是2^B = bucket数量
noverflow	overflow bucket的大约数量
hash0	哈希函数的种子值
buckets	指向bucket数组的指针
oldbuckets	在map扩容阶段指向前一个bucket数组的指针
nevacuate	在map扩容阶段充当**扩容进度计数器**，所有下标号小于nevacuate的bucket都已经完成了数据排空和迁移操作
extra	可选字段。如果有overflow bucket存在，且key、value都因不包含指针而被内联（inline）的情况下，这个字段将存储所有指向overflow bucket的指针，保证overflow bucket是始终可用的（不被GC掉）

极客时间

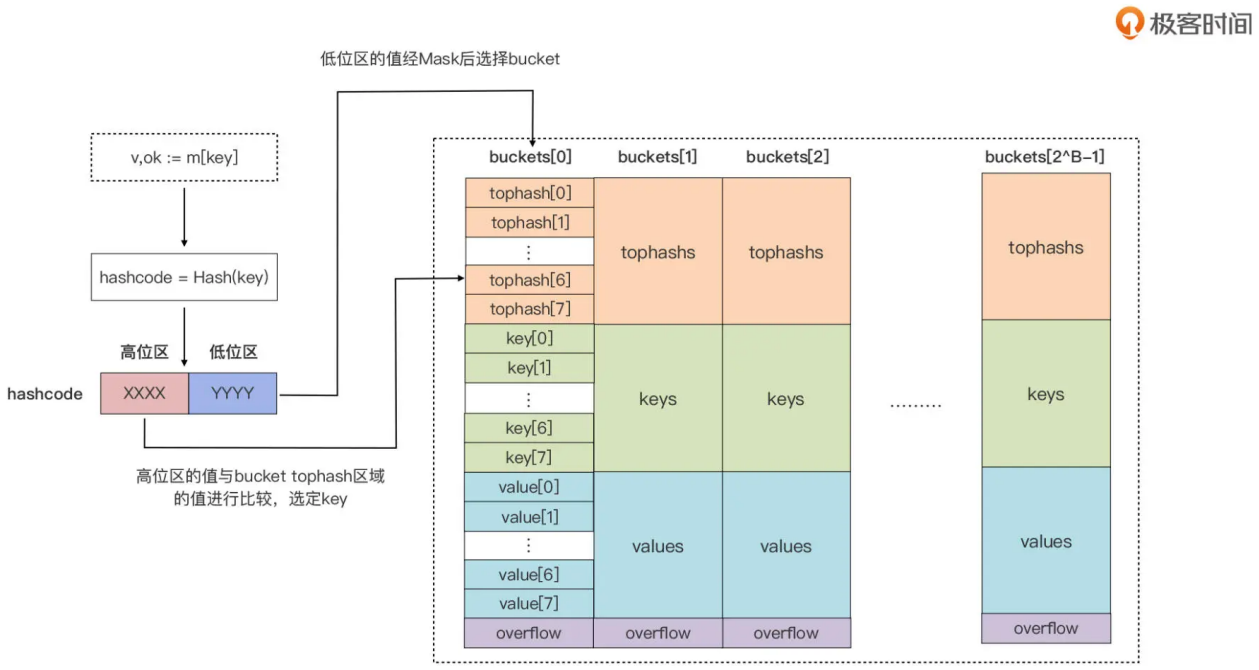
真正用来存储键值对数据的是桶，也就是 bucket，每个 bucket 中存储的是 Hash 值低 bit 位数值相同的元素，默认的元素个数为 BUCKETSIZE（值为 8，在 $GOROOT/src/cmd/compile/internal/gc/reflect.go 中定义，与 runtime/map.go 中常量 bucketCnt 保持一致）。

当某个 bucket（比如 buckets[0]) 的 8 个空槽 slot）都填满了，且 map 尚未达到扩容的条件的情况下，运行时会建立 overflow bucket，并将这个 overflow bucket 挂在上面 bucket（如 buckets[0]）末尾的 overflow 指针上，这样两个 buckets 形成了一个链表结构，直到下一次 map 扩容之前，这个结构都会一直存在。

从图中我们可以看到，每个 bucket 由三部分组成，从上到下分别是 tophash 区域、key 存储区域和 value 存储区域。

tophash 区域

当我们向 map 插入一条数据，或者是从 map 按 key 查询数据的时候，运行时都会使用哈希函数对 key 做哈希运算，并获得一个哈希值（hashcode）。这个 hashcode 非常关键，运行时会把 hashcode“一分为二”来看待，其中低位区的值用于选定 bucket，高位区的值用于在某个 bucket 中确定 key 的位置。我把这一过程整理成了下面这张示意图，你理解起来可以更直观：



因此，每个 bucket 的 tophash 区域其实是用来快速定位 key 位置的，这样就避免了逐个 key 进行比较这种代价较大的操作。尤其是当 key 是 size 较大的字符串类型时，好处就更突出了。这是一种以空间换时间的思路。

key 存储区域

接着，我们看 tophash 区域下面是一块连续的内存区域，存储的是这个 bucket 承载的所有 key 数据。运行时在分配 bucket 的时候需要知道 key 的 Size。那么运行时是如何知道 key 的 size 的呢？

当我们声明一个 map 类型变量，比如 var m map[string]int 时，Go 运行时就会为这个变量对应的特定 map 类型，生成一个 runtime.maptype 实例。如果这个实例已经存在，就会直接复用。maptype 实例的结构是这样的：

复制代码

```
type maptype struct {
    typ        _type
    key        *_type
    elem       *_type
    bucket     *_type // internal type representing a hash bucket
    keysize    uint8  // size of key slot
    elemsize   uint8  // size of elem slot
    bucketsize uint16 // size of bucket
    flags      uint32
}
```

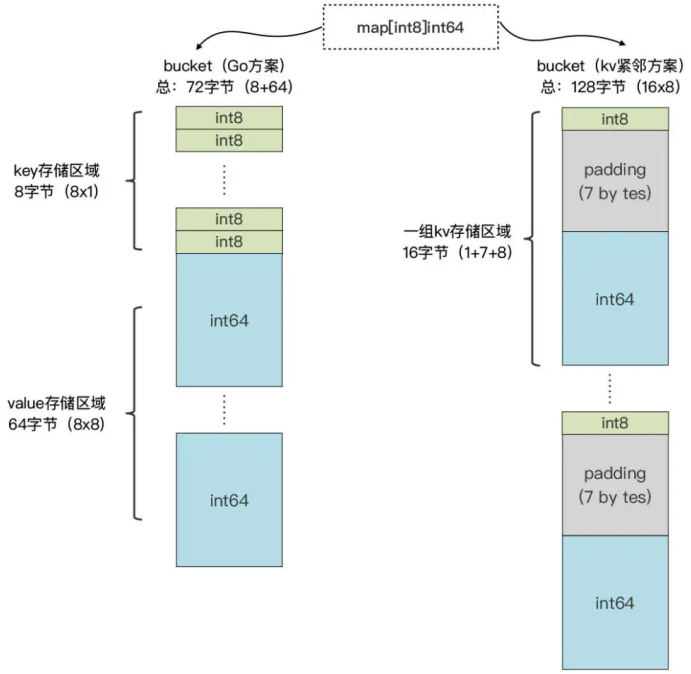
我们可以看到，这个实例包含了我们需要的 map 类型中的所有"元信息"。我们前面提到过，编译器会把语法层面的 map 操作重写成运行时对应的函数调用，这些运行时函数都有一个共同的特点，那就是第一个参数都是 maptype 指针类型的参数。

Go 运行时就是利用 maptype 参数中的信息确定 key 的类型和大小的。map 所用的 hash 函数也存放在 maptype.key.alg.hash(key, hmap.hash0) 中。同时 maptype 的存在也让 Go 中所有 map 类型都共享一套运行时 map 操作函数，而不是像 C++ 那样为每种 map 类型创建一套 map 操作函数，这样就节省了对最终二进制文件空间的占用。

value 存储区域

我们再接着看 key 存储区域下方的另外一块连续的内存区域，这个区域存储的是 key 对应的 value。和 key 一样，这个区域的创建也是得到了 maptype 中信息的帮助。Go 运行时采用了把 key 和 value 分开存储的方式，而不是采用一个 kv 接着一个 kv 的 kv 紧邻方式存储，这带来的其实是算法上的复杂性，但却减少了因内存对齐带来的内存浪费。

我们以 map[int8]int64 为例，看看下面的存储空间利用率对比图：



极客时间

你会看到，当前 Go 运行时使用的方案内存利用效率很高，而 kv 紧邻存储的方案在 map[int8]int64 这样的例子中内存浪费十分严重，它的内存利用率是 72/128=56.25%，有近一半的空间都浪费掉了。

另外，还有一点我要跟你强调一下，如果 key 或 value 的数据长度大于一定数值，那么运行时不会在 bucket 中直接存储数据，而是会存储 key 或 value 数据的指针。目前 Go 运行时定义的最大 key 和 value 的长度是这样的：

复制代码

```
// $GOROOT/src/runtime/map.go
const (
    maxKeySize  = 128
    maxElemSize = 128
)
```

map 扩容

我们前面提到过，map 会对底层使用的内存进行自动管理。因此，在使用过程中，当插入元素个数超出一定数值后，map 一定会存在自动扩容的问题，也就是怎么扩充 bucket 的数量，并重新在 bucket 间均衡分配数据的问题。

那么 map 在什么情况下会进行扩容呢？Go 运行时的 map 实现中引入了一个 LoadFactor（负载因子），当 **count > LoadFactor * 2^B** 或 overflow bucket 过多时，运行时会自动对 map 进行扩容。目前 Go 最新 1.17 版本 LoadFactor 设置为 6.5（loadFactorNum/loadFactorDen）。这里是 Go 中与 map 扩容相关的部分源码：

复制代码

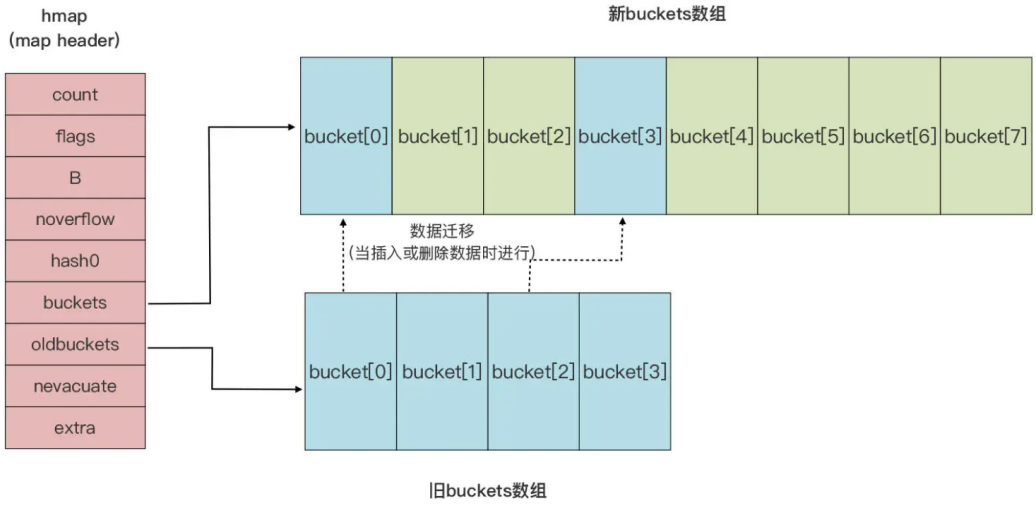
```
// $GOROOT/src/runtime/map.go
const (
   ... ...

   loadFactorNum = 13
   loadFactorDen = 2
   ... ...
)

func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
   ... ...
   if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets
      hashGrow(t, h)
      goto again // Growing the table invalidates everything, so try again
   }
   ... ...
}
```

这两方面原因导致的扩容，在运行时的操作其实是不一样的。如果是因为 overflow bucket 过多导致的“扩容”，实际上运行时会新建一个和现有规模一样的 bucket 数组，然后在 assign 和 delete 时做排空和迁移。

如果是因为当前数据数量超出 LoadFactor 指定水位而进行的扩容，那么运行时会建立一个**两倍于现有规模的 bucket 数组**，但真正的排空和迁移工作也是在 assign 和 delete 时逐步进行的。原 bucket 数组会挂在 hmap 的 oldbuckets 指针下面，直到原 buckets 数组中所有数据都迁移到新数组后，原 buckets 数组才会被释放。你可以结合下面的 map 扩容示意图来理解这个过程，这会让你理解得更深刻一些：



极客时间

map 与并发

接着我们来看一下 map 和并发。从上面的实现原理来看，充当 map 描述符角色的 hmap 实例自身是有状态的（hmap.flags），而且对状态的读写是没有并发保护的。所以说 map 实例不是并发写安全的，也不支持并发读写。如果我们对 map 实例进行并发读写，程序运行时就会抛出异常。你可以看看下面这个并发读写 map 的例子：

复制代码

```
package main

import (
    "fmt"
    "time"
)

func doIteration(m map[int]int) {
    for k, v := range m {
        _ = fmt.Sprintf("[%d, %d] ", k, v)
    }
}

func doWrite(m map[int]int) {
    for k, v := range m {
        m[k] = v + 1
    }
}

func main() {
    m := map[int]int{
```

```
        1: 11,
        2: 12,
        3: 13,
    }

    go func() {
        for i := 0; i < 1000; i++ {
            doIteration(m)
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            doWrite(m)
        }
    }()

    time.Sleep(5 * time.Second)
}
```

运行这个示例程序，我们会得到下面的执行错误结果：

复制代码

```
fatal error: concurrent map iteration and map write
```

不过，如果我们仅仅是进行并发读，map 是没有问题的。而且，Go 1.9 版本中引入了支持并发写安全的 sync.Map 类型，可以用来在并发读写的场景下替换掉 map，如果你有这方面的需求，可以查看一下 sync.Map 的手册。

另外，你要注意，考虑到 map 可以自动扩容，map 中数据元素的 value 位置可能在这一过程中发生变化，所以 **Go 不允许获取 map 中 value 的地址，这个约束是在编译期间就生效的。**下面这段代码就展示了 Go 编译器识别出获取 map 中 value 地址的语句后，给出的编译错误：

复制代码

```
p := &m[key]  // cannot take the address of m[key]
fmt.Println(p)
```

小结

好了，今天的课讲到这里就结束了。这一节课，我们讲解了 Go 语言的另一类十分常用的复合数据类型：map。

在 Go 语言中，map 类型是一个无序的键值对的集合。它有两种类型元素，一类是键（key），另一类是值（value）。在一个 map 中，键是唯一的，在集合中不能有两个相同的键。Go 也是通过这两种元素类型来表示一个 map 类型，你要记得这个通用的 map 类型表示：“map[key_type]value_type”。

map 类型对 key 元素的类型是有约束的，它要求 key 元素的类型必须支持"==“和”!="两个比较操作符。value 元素的类型可以是任意的。

不过，map 类型变量声明后必须对它进行初始化后才能操作。map 类型支持插入新键值对、查找和数据读取、删除键值对、遍历 map 中的键值数据等操作，Go 为开发者提供了十分简单的操作接口。这里要你重点记住的是，我们在查找和数据读取时一定要使用“comma ok”惯用法。此外，map 变量在函数与方法间传递的开销很小，并且在函数内部通过 map 描述符对 map 的修改会对函数外部可见。

另外，map 的内部实现要比切片复杂得多，它是由 Go 编译器与运行时联合实现的。Go 编译器在编译阶段会将语法层面的 map 操作，重写为运行时对应的函数调用。Go 运行时则采用了高效的算法实现了 map 类型的各类操作，这里我建议你要结合 Go 项目源码来理解 map 的具体实现。

和切片一样，map 是 Go 语言提供的重要数据类型，也是 Gopher 日常 Go 编码是最常使用的类型之一。我们在日常使用 map 的场合要把握住下面几个要点，不要走弯路：

不要依赖 map 的元素遍历顺序；

map 不是线程安全的，不支持并发读写；

不要尝试获取 map 中元素（value）的地址。

思考题

通过上面的学习，我们知道对 map 类型进行遍历所得到的键的次序是随机的，那么我想请你思考并实现一个方法，让我们能对 map 的进行稳定次序遍历？期待在留言区看到你的想法。

欢迎你把这节课分享给更多对 Go 语言 map 类型感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | 同构复合类型：从定长数组到变长切片

下一篇 17 | 复合数据类型：用结构体建立对真实世界的抽象

精选留言 (12)

写留言

罗杰

2021-11-17

还想深入研究 map 的小伙伴，可以去研究这些博客：

https://www.qcrao.com/2019/05/22/dive-into-go-map/
https://qcrao.com/2020/05/06/dive-into-go-sync-map/
https://draveness.me/golang/docs/part2-foundation/ch03-datastructure/golang-hashmap/

展开

3

ddh
2021-11-17

把key存到有序切片中，用切片遍历

展开

共 1 条评论 >

2

2021-11-17

可以参考java的LinkedHashMap,能实现插入有序或者访问有序，就是使用额外的链表来保存顺序。
go 中可以基于container/list来实现。github上现成的功能。
https://github.com/elliotchance/orderedmap

展开

2

用0和1改变自己
2021-11-17

可以用一个有序结构存储key,如slice,然后for这个slice,用key获取值。资料来源至：https://go.dev/blog/maps

展开

2

AlexWillBeGood
2021-11-18

老师，有一点我不明白，为什么kv 紧邻存储的时候一定需要Padding ?

共 2 条评论 >

1

酥宝话不多
2021-11-17

这里 p 拿到的不是地址，而是 key 对应的 value

展开

高雪斌
2021-11-17

```
func doIteration(m map[int]int) {
    mu.RLock()
    defer mu.RUnlock()

    keys := []int{}...
```

展开

共 1 条评论 > 1

Geek_0b92d9
2021-11-17

老师好。我是 go1.16.2 版本

var m map[string]int 定义后， 打印 m，是 map[]，不是 nil

共 1 条评论 > 1

return
2021-11-17

老师讲的太好了，

展开

aoe
2021-11-17

使用链表存储可以得到有序的Map。有了描述符，再也不用担心传递性能问题了！原来多个bucket是为了降低Hash冲突，虽然一个bucket和多个bucket在查找Key时时间复杂度都是O(1)，但一个Bucket遇到Hash冲突的可能性要比多个高出很多。

展开

1

liaomars
2021-11-17

老师：函数类型、map 类型自身，以及切片类型是不能作为 map 的 key 类型的。
这个是不是可以理解成：函数类型，map类型，切片类型是不能做为key的？

共 1 条评论 >

Badb0y
2021-11-17

可以对KEY排序后再遍历

展开